SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 12, 655-667 (1982)

The Kernel of the EMAS 2900 Operating System

D. J. REES
Department of Computer Science, Edinburgh University, Mayfield Road, Edinburgh EH9 3JZ, Scotland
AND
P. D. STEPHENS
Edinburgh Regional Computing Centre, Mayfield Road, Edinburgh EH9 3JZ, Scotland

SUMMARY

The role of the kernel of the operating system EMAS 2900 and the implementation of its functions are described in some detail. The significance of local scheduling policies and their implications on the design of the kernel are discussed with particular reference to paging management and scheduling control. It is shown that the concept of local and global control of resources can lead to a considerable simplification in the structure of an operating system kernel. The resulting operating system, EMAS 2900, provides interactive time-sharing services very effectively and efficiently to a large computing community.

KEY WORDS Operating system Multi-access Scheduling policies

INTRODUCTION

EMAS 2900 is a multi-access time-sharing operating system for the ICL 2900 series of computers which was developed by the Department of Computer Science and the Edinburgh Regional Computing Centre in Edinburgh University. The development was a re-implementation of the EMAS system, which ran on the ICL System 4-75 computer, using the same underlying philosophy but taking into account the experience of several years' use and new insights which had resulted. As with any large system, the EMAS implementation had become more difficult to maintain as time went by and as the original team left the scene. A significant goal was therefore to achieve a simplification of the system in order to facilitate future improvements and to postpone the point when complexity escalation made further changes difficult. There was also a strong desire to exploit the architecture of the 2900 series in relation to multi-access systems. An overall view of this project has been described by Stephens et al.[1] The purpose of the present paper is to describe the kernel of the system in some detail, in particular the organization of the virtual memory control and scheduling.

One of the most important design concepts in both EMAS systems has been that of process-local page replacement policies. (The term 'process' is here used to signify the operation of a virtual machine consisting of a virtual processor, i.e. one which provides access to the non-privileged instruction set plus various system services, operating within a virtual address space.) Overall performance in terms of interactive response and efficiency of use of the hardware has vindicated this choice in comparison with systems using global policies. Theoretical studies by Denning also bear out the wisdom of this choice.[2] The recognition of the significance of this policy motivated the

0038–0644/82/070655–13$01.30 *Received 17 November 1981*

main difference in structure between the two EMAS systems. Whereas in EMAS the implementaiton of the local policies was intermingled with the rest of the resident kernel, in EMAS 2900 a very clear separation has been made between local policy controllers and controllers of global resources. Each process contains an incarnation of a local controller whose function is to control those resources which have been allocated to that process from the global scheduling controller. This notion of completely separated local controllers has also been utilized with very great benefit in the design of the communications subsystem, described by Laing.[3]

The implementation of this policy of separation was facilitated by the organization of 2900 virtual address spaces. Each virtual address space of 2^{32} bytes is divided into two halves, a 'local' half unique to each process and a 'public' half common to all processes. In an EMAS 2900 process the local half contains all the programs and data used by that particular process together with an incarnation of the local controller, the director and the subsystem. The director is the innermost layer of software of a process and incorporates many of the local system services such as the file system services. The subsystem implements the next layer of the hierarchy which includes the basic command interpreter, editors, compilers, loaders, etc. The functions of both these modules remain essentially as they were in EMAS.[4, 5] As much as possible of this material is shared between processes using the standard sharing mechanisms of the system. The local controller code is shared but it was found convenient to compile this as a module of the kernel—this will be clarified later. The public half of the virtual address space contains the kernel of the system, i.e. the global controller, the message passing dispatcher, device handlers, etc.

The kernel thus appears in every process address space and runs in virtual mode, unlike most earlier hardware designs such as the 4-75 where it ran in real address mode. Switching between the current local space and the kernel does not involve switching virtual machines. Peripheral interrupts can be directed to an address in the kernel in the same virtual space, while interrupts such as page faults and local process time-outs can be taken directly by the local controller. There they can immediately be dealt with according to the resources that have been allocated to that process. The resources in question are primarily pages of physical storage and CPU time but there are also various other internally defined resources such as 'active memory' sections (described below) which also have to be controlled.

The page size of the 2900 series is 1K bytes but in the light of the implementors' experience on EMAS we decided that this would be too small, and in EMAS 2900 these pages were grouped together to form larger units. We had originally hoped to be able to experiment with different unit sizes but this proved to be infeasible owing to the difficulties of varying physical block formats on disc and magnetic tape. The unit size eventually chosen was the same 4K bytes that we had used on EMAS; this gave compatibility between the systems in addition to being what we felt was a sensible choice. 'Pages' hereinafter refer to these 4K byte multiple pages.

In order to share information, movement of pages is initiated and controlled globally but in response to requests from local controllers. A local controller is unaware of any sharing of pages which is taking place between processes. It makes requests for particular pages to be made available to it in main store which the global controller then fulfils as best it can. If the page is already being used by another process or is still in store from a previous usage the global controller can simply tell the requesting local controller where the required page is and allow it to continue.

Alternatively it may have to fetch the page from a disc with consequent delay. This is all transparent to the local controller, which simply requests the page and gets a reply. Similarly, when a local controller decides that the process it controls no longer needs a particular page, it tells the global controller and leaves the global controller to remove it. If the global controller knows that another process is still using the page it will not page it out. This method of operation in which the local controller is concerned only with its own process makes implementation of the local controller much easier and the result more reliable. The details of the global controller are described first; its organization reflects the two functions of paging control and scheduling of resource allocation. The former can be regarded as including disc and drum handlers but these will not be discussed here as they are relatively straightforward and not of any great originality. Each device driver or control function is a separate resident computation or process which runs uninterruptably to completion. These communicate with each other and with local controllers using a message passing system exactly as in the original EMAS.[6]

The whole of the operating system is written in IMP, the language used in Edinburgh University for most systems implementation work. This was described by Stephens.[7] The architecture of the 2900 series was designed very much with the use of high level language in mind (for instance, see Buckle[8]). In particular, the hardware defines a stack segment which can be used as the stack for IMP storage allocation and procedure calling protocols. A potential problem for an operating system kernel is the size of its internal arrays when the number of users and processes is likely to vary considerably over a range of computers (such as the 2900 series). This has been overcome in the EMAS 2900 kernel by making use of virtual space. Each array that may need to be extended is mapped into a separate segment. Physical store pages claimed from the free page-frame list can be added onto the end of the segment when required by amending the appropriate page table.

Extensive monitoring of the performance of the system was carried out during its development and since it has entered service, both by internal measurement and by external measurement using the Edinburgh Remote Terminal Emulator (ERTE).[9] This proved to be of immense value in determining the best approach to certain aspects of the design and indicated the most fruitful developments to pursue.

PAGING CONTROL

A fundamental feature of EMAS is the way in which virtual memory is used. Files are mapped into virtual memory such that when a particular virtual address is accessed the corresponding item in the file is referenced. The action of creating the mapping between a file and an area of virtual memory, termed 'connection', is purely a logical operation and no file data is transferred at this time. Physical movement of file data is only initiated when a page fault occurs. This is the 'one-level store' concept implemented in many systems. Furthermore a user normally does not know on which disc his files are stored. The allocation of disc space is handled entirely behind the scenes by the file system. A file resides on disc storage as a set of disc 'sections' (sometimes termed 'extents' elsewhere) as a matter of convenience for the file manager. The connection mapping therefore consists of a table containing the

locations on disc of all the sections of the files which are currently connected. This table forms part of the local controller and will be described in detail later. The significance to paging control lies in this division into sections.

The file is the unit of shareability to EMAS users and all files are potentially shareable between any number of processes. Since files may be very large, it is more convenient to use the section as the basic unit which the global controller handles rather than the file. When a local controller requires a file page for its process, it must first ensure that the section within which that page lies is 'active'. 'Activating' a section takes the form of a request from the local controller to the global controller specifying the disc address and length of the section together with a 'new page' mask. The global controller then sets aside an appropriate data structure for this section (an 'active memory table entry') and allocates a logical section number, known as an 'active memory table index' (*amtx*), which is used thereafter to identify the section. Individual page requests specify the *amtx* and a page-within-section number. The new page mask allows the local controller to specify that certain pages in the section have never had data written into them. This allows the global controller to avoid making a physical transfer from backing store when such pages are requested. A free store page-frame is simply allocated and cleared to zero. Figure 1 includes the data structure for such an activated section.

The collection of AMT entries for all the active sections forms a dictionary through which sharing can be controlled. It is organized as a hash table using linked lists of entries and with the disc address as the key. The disc address could have been used instead of the *amtx* value to identify the active section but search time is saved by using the latter. Sections shared between processes appear having a 'users' count greater than one. The 'outs' field contains a count of the number of page-out transfers belonging to this section currently in progress. When the 'users' and 'outs' counts are both zero, the section can be removed from the AMT dictionary. The pointer field shown indicates a collection of entries, one per page in the section, which record the current status of each page. This area is allocated within a single global array, the 'Active Page Table', and since sections may have different lengths, a dynamic allocation scheme is used.

The status information for each page consists of two flag bits and a pointer field. The first flag bit is the 'new' bit distributed from the activation mask and the second indicates whether there is a copy of the page on drum storage. Where the hardware installation does not have drum storage this latter bit is always zero. The pointer field contains a null value if the page only resides on disc, a pointer to a drum table entry if the on-drum bit is set, or a pointer to a store table entry otherwise. Since drum table entries correspond one-for-one with page-frames on drum, the pointer also denotes the position of the page on drum. A drum table entry contains a pointer to a store table entry if there is also a copy of the page in store, or a null value otherwise. This is also illustrated in Figure 1.

Each store table entry contains the physical address of the page, a count of processes currently sharing the page, a set of flags and various links. The flags indicate whether a backing store transfer is in progress to or from this page, whether the page has been modified since being paged in (and thus requiring to be written out in due course rather than discarded) and whether the page is 'recaptureable'. A page is said to be recaptureable when it has been discarded onto the free page-frame list and before it has been used for anything else. In these circumstances, a page-in request for the page

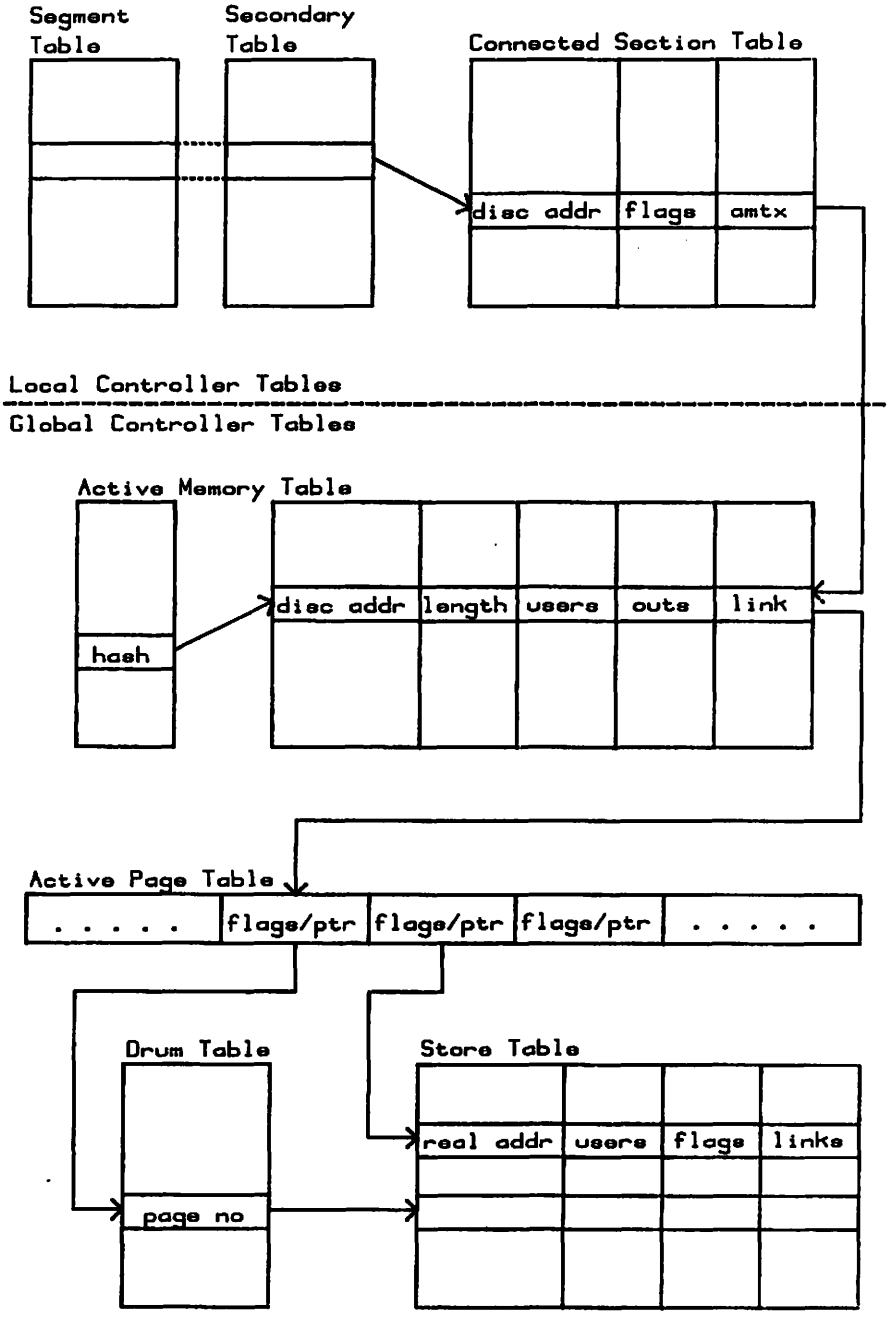

Figure 1

will recapture the page-frame from the middle of the free list and avoid the need for a transfer. This mechanism has a very significant effect on the performance of the system: 30 per cent of all page-in requests are satisfied by recapture under heavy load, and a higher percentage under light load.

The free page-frame list is formed from the store table by linking together those entries not in use. To enable page-frames to be recaptured from the middle of the free list it is constructed with both forward and backward links. A third link in each entry is used in two different ways. When the page is recaptureable, it is used to refer back to the AMT entry so that page-frame numbers can be removed therefrom when the page-frame is claimed for a different purpose. When the page is in use and a page-in transfer has been initiated, it is used to point to a list of processes waiting for the page to arrive in store. This is referred to as a page-in-transit list and takes the form of messages that will be forwarded to the processes when the transfer has been completed.

The sequence of operations which takes place when a page-in request is made can be summarized as follows. If a page-frame is already allocated for this file page, a reply can be generated immediately, except for the case when a page-in transfer is in progress. In this case an additional entry is made in the page-in transfer list. If a page-frame has not been allocated to the file page requested, a request is made for a free one. When a page-frame has been allocated, the AMT entry is examined. If the page is marked as new the page is cleared and a reply sent. The 'modified' marker in the store table is also set to ensure that the cleared page is written out subsequently. If the page is not new, the backing store position of the page is determined and a transfer initiated as appropriate. In the case of page-in transfers, the replies from the device manager are normally sent direct to the local controller which orginated the request rather than via the paging manager; this results in a worthwhile reduction in overhead. Only when a drum read fails is a reply sent back to the paging manager. In this case, the transfer is requested from the disc site instead and the drum site is marked as 'bad'. In the case of disc read failure, the local controller which originated the request is informed and an error is signalled to the process.

For page-out requests the local controller assumes that the paging manager will be able to perform the task; consequently no reply is made from the paging manager. This simplifies the local controller considerably as it enables it to return all its notional page resources to the scheduling manager as soon as all the page-out requests have been made. This has the effect of maximizing the overlap between a process being paged out and a new process being paged in. The only slight disadvantage is that very long queues of backing store transfers can build up.

If a page for which a page-out request has been made has more than one user, the page will remain in store and no transfer will be carried out, although information on whether the page was modified will be recorded for later use. If the page only had one user it can immediately be paged out. Here another major simplification from EMAS can be observed: the drum is now regarded as a higher-speed cache of active disc pages, so that when a page is transferred out, it is transferred to both drum and disc, not just to the drum. Subsequent paging-in will take place from the drum site, except in the rare case of drum failure. The very great advantage is that when a section is deactivated, only housekeeping actions are required and no transfers have to be made from drum to disc. Furthermore, in the case of a system crash, the disc file site will be more up-to-date than if many of the modified pages are only on drum. Clearly there is more overhead in setting up both transfers but this has to be balanced against the greater simplicity.

Pages which were not modified will not be transferred out again, with the exception that if a valid copy of the page does not yet exist on drum, a transfer out to drum alone

is made. When all transfers out have been completed the page-frame is returned to the free list but is marked as recaptureable.

SCHEDULING CONTROL

One of the fundamental scheduling problems in time-sharing paged systems is that of thrashing, the phenomenon of continual page-faulting from processes that are unable to acquire an adequate number of pages in store. In EMAS 2900, as in EMAS, this is avoided by carefully controlling the number of processes in the 'multiprogramming set', i.e. those processes allocated main store and potentially able to run on the CPU. (The period during which a process is in the multiprogramming set is referred to as a 'store residence'.) By making an estimate of the number of pages each process requires to run efficiently, the scheduler can ensure that the processes it decides to allow into the multiprogramming set do not overload the available store. The estimate of how much store a process will require the next time it enters the muliprogramming set is made from its previous behaviour. This is a good indicator but clearly a process will occasionally begin to exhibit different characteristics. For this reason the scheduler is designed to be adaptive and changes of behaviour are dealt with automatically. The adaptation is derived from a table of categories through which processes migrate. Each category defines a combination of store and CPU-time requirement together with a priority, a set of transition indicators and various other data. The system attempts to be fair in the sense that categories which define lower resource requirements are allocated higher priorities and vice versa. The transition indicators control the route through the categories which a process takes as a consequence of variations in its resource requirements. A typical category table is shown in Table I.

Table I

Category	Pages	Time	Priority	More pages	More time	Less pages	Run q 1	Run q 2	Strobe
1	20	4	1	2	5	1	1	1	0
2	32	4	2	3	6	1	1	1	0
3	64	4	2	4	7	2	1	1	0
4	128	4	3	4	8	3	1	1	0
5	20	24	2	6	9	5	1	2	0
6	32	24	2	7	10	5	1	2	0
7	64	24	3	8	11	6	1	2	0
8	128	24	4	8	12	7	1	2	4
9	20	64	3	10	9	9	2	2	0
10	32	64	4	11	10	9	2	2	8
11	64	64	4	12	11	10	2	2	8
12	128	64	5	12	12	11	2	2	8

The category table shown is a fairly basic one that has undergone no tuning to fit a particular installation. Being table-driven, the scheduling is very easy to change but the system-wide ramifications of local scheduling changes are not always immediately apparent. The regular use of the ERTE remote terminal emulator system was of particular value in this respect. One modification which is normally present is to incorporate into a table a disjoint set of categories and their transitions suitable for jobs submitted to a batch stream. EMAS 2900 has facilities which allow a number of

processes to be created as batch processing streams but these need different priorities so as not to interfere with the response of interactive processes. When a process is placed in a certain category the scheduler tells the local controller how many pages it may acquire on behalf of the process and the number of time-slices of CPU it may consume (the 'pages' and 'time' columns of the table). The local controller can then allow the process to proceed within these limits. If these resources are not sufficient, the local controller is obliged to request to be rescheduled. In principle this means that that process will be removed from the muliprogramming set in order to give other processes a chance to make progress. In asking to be rescheduled, the local controller says which resource it has run out of and how much of the other resource it had used by this stage. This allows the scheduler to adapt the category of the process to one which will be more suitable. For example, if the process had run out of time, the scheduler would assign it to a new category that had a greater allocation of CPU time. This category is indicated in the 'more-time' column of the category table. In addition, if there is a category with fewer pages but still with more pages than the process had used when it ran out of time, that category would be chosen instead in order to minimize overall resource allocation and to maximize the priority of the process.

Local controllers endeavour to avoid exceeding their page allocations by examining the page usage markers in their page tables at certain times. If this indicates that a page has not been accessed recently then it can be paged out. This activity, known as strobing, is controlled by the category in which the process resides because its effectiveness varies considerably depending on the characteristics of the process and because strobing is an expensive operation.

The priority given to a process from its category is used by the scheduler to control when it is to be admitted to the multiprogramming set. The scheme is not pre-emptive but uses a 'priority ratio' table which indicates the priority of process that the scheduler should load next. Thus higher priority processes will be admitted frequently but the lower priority processes can never be entirely excluded from making forward progress. Normally when a process changes to a lower priority category it is removed from the multiprogramming set and requeued. However it is allowed to change priority and remain in the multiprogramming set if none of the waiting processes have a higher priority than the one that has just changed its category. This tends to occur when the system is lightly loaded and avoids the cost of paging a process out and in again unnecessarily.

There are two stages to admitting a process to the multiprogramming set. The first is to allocate pages for it and to page in the local controller stack; the rest of the local controller is shared and already resident. The second is to allocate the pages for the process itself and activate the local controller, which then passes control to the process and deals with its page-faults. Since the local controller stack is only three pages long, an allocation for this can almost always be made well ahead of sufficient pages becoming available for the remainder of the allocation. The scheduler then immediately pages in the local controller stack using exactly the same mechanisms and facilities, described above, that local controllers themselves use. The pages are thus very likely to be in store by the time that the full allocation is available, and the potential inefficiency of two stages is largely avoided.

In EMAS and in the initial version of EMAS 2900, a 'working set reloading' scheme (also known as 'preloading') was used.[10] The working set of each process was

recorded at the end of a store residence and these pages were automatically paged in again at the start of the next store residence. On the 4-75 this scheme was very beneficial in improving response. On the 2900s it proved to be rather less so for various reasons. Firstly, on machines with drum storage there was very little performance gain since the drums were so fast. Secondly, the inaccuracy of preloading, i.e. those pages reloaded but never actually accessed, was for some unexplained reason also rather higher on the 2900s than it had been on the 4-75, which resulted in more wasted effort. Thirdly, an inaccurate preload destroyed potentially recaptureable pages—a situation that did not occur on the 4-75 where recapturing was not implemented. In the absence of drum storage the benefits of preloading might have been expected to reassert themselves but the substitution of large main stores meant that page recapture tended to be much more effective and to perform much the same function.

The method of totting up the process page allocations so as not to overload the available store is effective but has drawbacks. One is that a shared page will be accounted as a page for each process using it although only one copy will reside in store; this is trivially overcome. A more serious drawback arises from the fact that any particular process is liable not to be using the whole of its allocation at any point in time, a form of internal fragmentation. This disadvantage can be mitigated by overallocating to a limited extent, say by 20 per cent. Since requests for store pages cannot always be immediately satisfied (a previous process may still be being paged out, as mentioned above), a queuing mechanism for requests already exists. With overallocation, more queuing takes place but the overall gain is noticeable. However there is now the possibility of deadlock arising: if no more page-frames are expected to be released when page-outs complete and all the processes are requesting a further page (via the paging manager) then a deadlock situation has arisen. As long as the overallocation is kept reasonably small such deadlocks only occur rarely. When one does occur the solution we adopted was to detect that one has occurred in the store page allocation routine and arbitrarily to force one local controller to page out its process. This will release pages for the remining processes to proceed. This is admittedly somewhat messy but effective. A final imperfection of totting up is that unless some extra action is taken the store may become dominated by large processes, even though they have low priority. This would have the effect of limiting the number of small interactive processes in the multiprogramming set and correspondingly limit their overall rate of response. The problem can be overcome by arbitrarily limiting low priority processes to a proportion, such as half, of the total store.

When a process wishes to wait for some external event such as console input, the local controller has to request the scheduler to suspend it. If there is sufficient store available the scheduler may allow the process to 'snooze' in store without being paged out, but otherwise the local controller will have to page its process out before suspending. An intermediate form of snoozing has also been implemented which consists of retaining just the local controller stack. This occupies much less space and still improves response to the user when the process is unsuspended.

The message-passing scheme adopted in EMAS 2900 is very similar to that described in EMAS[6] but a certain amount of extra flexibility has been built in for paged processes. Basically, messages are addressed to a particular service number but any one service routine may accept several service numbers. Service numbers can also be 'inhibited'. This inhibits the message dispatcher from delivering messages on that

number. A typical usage of this facility is when a device handler cannot deal with any more transfer requests for the moment.

Dispatcher priorities amongst processes in the multiprogramming set use the '*run q*1' and '*run q*2' fields from the category table. The dispatcher maintains two run queues for paged processes (they are, in fact, message queues and the message dispatcher is one and the same thing as the paged process dispatcher). Processes on the first have pre-emptive priority over those on the second and the queue that a process goes on is defined by '*run q*1' and '*run q*2'. '*run q*1' indicates the queue that the process is to go on during its first time-slice in the round-robin CPU-allocation scheme used for the processes in the multiprogramming set. '*run q*2' indicates the queue it should go on during the second and subsequent time-slices of its residence. This control allows highly interactive processes to be given favoured treatment and improves their response times significantly. Processes which only ever go on the second queue (i.e. *run q*1 = *run q*1 = 2) still make reasonable forward progress since the processes on the first queue are such that they do not hog the CPU. Most of them do not even use up the complete time-slice.

THE LOCAL CONTROLLER

The essential functions of the local controller have been described above in the contexts of paging and scheduling control. The organization of its private data structures and other details remain to be described.

The incarnation of the local controller for each virtual process consists of shared code which is common to all local controllers and a private data segment suitably protected from user access by the 'rings of protection' mechanism provided by 2900 series hardware. This segment is used to hold both the local segment table for the process and the stack which is required for running the local controller as an IMP program.

Since a local controller is intimately associated with the global controller, it was found convenient to compile the local controller code as a procedure within the kernel rather than as a separate entity. This implies that whenever the local controller procedure is called the stack has to be switched from the one on which the kernel is running to the local controller's own stack. 2900 series architecture provides a mechanism—the 'outward call'—to do this.[8] The great advantage of this scheme is that all the data structures required by the local controller can be allocated by the normal IMP mechanisms, and on a stack which can be paged out when the process is not in the multiprogramming set. A subsidiary advantage is that the local controller can access global controller data structures just as global variables, even though they are on a different stack. Philosophically this is perhaps undesirable but it makes life very much simpler. Since outward calls are expensive operations they are only used on the initial entry to each incarnation of local controller. Local controller thereafter exits to kernel with a software generated interrupt and is then re-entered at the point of interruption with all its data intact.

In order to access a file it is mapped into virtual space by being 'connected'. (This is one of the functions of the file system.) To make a connection the director writes mapping information into the local controller's data structures. A segment is made accessible to the director for this purpose; the local controller tells the director where the data structures are within this segment on start-up. This requires co-operation

between the two layers of the system, local controller and director, to avoid inadvertent misuse but has not proved to be a problem from the point of view of reliability. The data structures which define connection mappings consist of a table known as the 'secondary segment table' (parallel to the local segment table), each entry of which points to a set of records in the 'connected section table'. Each of these records describes a section of storage on disc.

The description of each section of storage contains the disc address of its start, its length and a field which is used to hold either the 'new' bits for pages in the section when it is not active, or the *amtx* active memory table index received from the global controller when it is active. The remaining tables maintained by the local controller are concerned with active sections. For each active section, those pages within it which are also active, i.e. in use by this process, must be remembered. Furthermore, an abbreviated history of the usage of active sections must also be maintained. The structure of the tables went through several iterations during local controller development. These iterations were entirely local to the local controller and therefore posed no interface problems with the global controller. This was another benefit of the modularization into local and global controllers. Originally, linked lists were extensively used but these proved to take more space than was desirable and so bit maps, which were more compact, were substituted.

The main bit map consists of an array of 64-bit words, each word of which relates to some segment and each bit to a page within a segment. One of these words is set aside when any section within a segment is first activated. This allows up to 32 segments to be active at any one time, a limit which has proved more than adequate in practice. Nevertheless, as the context of a process changes this limit would be exceeded through old segments still being active although not in current use. The local controller therefore attempts to maintain a working set of active segments and to deactivate sections within segments that are no longer within that set.

The construction of page tables for each of the segments of local virtual space in use is a further function of the local controller. For this purpose the local controller is allowed to claim physical page frames directly from the global allocator instead of indirectly through the paging manager, but they still must be accounted for in the local controller's notional page allocation. Several maximum sized page tables can be packed into one page and many more when the segments are of the typical small size.

The remaining main function of the local controller is to handle communications with the director of the process. Interprocess messages are forwarded by the local controller, for instance, together with an inevitably growing list of specialized services that have been thought to be desirable as the system has developed.

MULTIPROCESSOR KERNELS

The 2900 series provides for dual processor configurations and there were suggestions that multiple processors would be available in due course. The attractions of more than one CPU for improved reliability are obvious and the software design has always had multi-CPU configurations in mind. Since the attraction of multiprocessors was seen as the avoidance of interruptions of service rather than obtaining better response, we aimed for an entirely symmetric arrangement of processors. All processors would execute the same dispatcher and be able to handle all interrupts. This was essential for

the system to live through the various halts, power-offs and loops that seem to occur nearly as often as properly signalled hardware errors.

The actual extension of the kernel to handle several processors was very simple. It is necessary to ensure that a local controller, or its user process, is being executed by only one processor at any time and this was done by an extension of the inhibit mechanism described above. A similar extension applied to kernel services which are performed by resident processes. The queuing/dispatching operations were protected by a semaphore.

Peripheral and clock interrupts are broadcast to all CPUs and all try to read and clear the interrupt flag register. Only one will succeed; it deals with the interrupt, which involves turning it into a service request and queuing it. The other processors resume their interrupted tasks. The remaining problem area was the three main tables, Store, Drum and Active Memory: these are all accessed by more than one process and must therefore be protected by semaphores.

Semaphores are claimed and released by the special 2900 operations provided; claiming or releasing a semaphore also invalidates the slave store in the processor and forces any unfinished write cycles to complete. If a semaphore clash is detected a wait routine is entered until the semaphore has been freed. If the semaphore is not freed within one second it is assumed that a processor has failed or stopped and recovery operations commence. To avoid a deadly embrace any processor is allowed to claim only one semaphore; this restriction necessitated some re-ordering of operations in the paging manager.

The multi-processor kernel described above was operational within 2 weeks of the delivery of the dual 2972, although some preliminary testing had been done on a dual 2970 at Southampton University. The improved reliability was immediately obvious. The time spent waiting for semaphores was around 1 per cent of total CPU time.

In the current version of the kernel some improvements have been made. The global inhibiting of the device servers is rather inefficient so they have been exempted from the scheme and operate their own semaphores at the device level. Thus one processor can be starting a transfer on one unit while another is dealing with an interrupt from another. This licence has also been extended to the paging manager and communications controller.[3] The broadcasting of interrupts has also ceased—this is because interrupts clear the slave stores and consequently a null interrupt is expensive. The current, less elegant, scheme is to route clock interrupts to one processor and device interrupts to the other. The idle loop is used for each processor to check and cover for the other. If interrupts appear to be missing the system reverts to the original method. These changes have improved response and reduced semaphore wait time to around 0·2 per cent of total CPU time.

The aim of increased reliability has been achieved with the system continuing through a substantial majority of processor failures as a single processor configuration.

CONCLUSIONS

The exercise of transporting EMAS to the 2900 series hardware has been a conspicuous success as far as its users are concerned. It has proved much more effective than the manufacturer's operating system in a University environment, mainly, we believe, because our objectives were clear and because we did not aim to produce a system that was all things to all men. The opportunity to rethink the design

of the kernel after a number of years' experience with EMAS was invaluable and the resulting design has improved on the original in almost every respect. The simplicity and elegance that we sought has largely been achieved. Though one can always think of ways one could do it even better next time, by and large we are satisfied with the outcome of our efforts.

ACKNOWLEDGEMENTS

The implementation of a large system such as EMAS 2900 is very much a team effort. The constitution of our team changed over the period of the development but the hard core of the implementors who stuck to the task deserve special mention for their efforts and constant availability to discuss ideas and develop the EMAS philosophy. They were J. K. Yarwood and N. H. Shelness together with the able assistance of W. A. Laing, R. R. McLeod and F. Stacey.

REFERENCES

1. P. D. Stephens, J. K. Yarwood, D. J. Rees and N. H. Shelness, 'The evolution of the operating system EMAS 2900', *Software—Practice and Experience*, **10**, 933–1008 (1980).
2. P. J. Denning, 'Working sets past and present', *Purdue University Report CSD-TR-276*, 1978.
3. W. A. Laing, 'Communications control in the operating system EMAS 2900', *presented at IUCC*, September 1981.
4. D. J. Rees, 'The EMAS Director', *Computer Journal*, **18**, 122–130 (1975).
5. G. E. Millard, D. J. Rees and H. Whitfield, 'The standard EMAS subsystem', *Computer Journal*, **18**, 213–219 (1975).
6. H. Whitfield and A. S. Wight, 'EMAS—the Edinburgh multi-access system', *Computer Journal*, **18**, 331–346 (1973).
7. P. D. Stephens, 'The IMP language and compiler', *Computer Journal*, **18**, 131–134 (1975).
8. J. K. Buckle, *The ICL 2900 Series*, Macmillan, London, 1978.
9. J. C. Adams, W. S. Currie and B. A. C. Gilmore, 'The structure and uses of the Edinburgh remote terminal emulator', *Software—Practice and Experience*, **8**, 451–459 (1978).
10. J. C. Adams: 'Performance measurement and evaluation of time-shared virtual memory systems', *Ph.D. Thesis*, Edinburgh University, 1977.